4
Algoritmos Propostos

Neste capítulo serão descritos os algoritmos propostos para melhorar o desempenho de sistemas com ordenação no início e sistemas híbridos com ordenação no início e no fim.

Devido à distribuição desigual de esforço sobre a tela, um dos maiores desafios de sistemas com arquitetura com ordenação no início é dividir a tela em ladrilhos de igual carga para que nenhum nó de renderização fique ocioso.

Após a definição do problema de balanceamento de carga, será apresentado um algoritmo que utiliza os tempos de renderização dos ladrilhos em quadros anteriores para inferir uma boa partição da tela para os quadros subseqüentes. Além disso, será apresentada uma estratégia de distribuição de ladrilhos entre os nós de renderização que procura maximizar o uso dos recursos gráficos do agrupamento de PCs [27].

Finalizando o capítulo, será descrito um algoritmo de partição do modelo entre os nós de renderização para a arquitetura híbrida com ordenação no início e no fim.

4.1
O Problema de Balanceamento de Carga

A utilização efetiva do poder computacional das unidades de processamento disponíveis em um sistema paralelo é o grande desafio da área de computação paralela e distribuída. O sucesso nesse desafio depende em grande parte do modo como os dados ou funções são particionados entre os processadores do sistema. Devido a isto, um dos objetivos de sistemas paralelos é fazer uma partição balanceada do trabalho disponível entre os processadores.

Especificamente em sistemas de renderização distribuídos que visam propiciar interatividade em tempo real, é necessária a manutenção de uma

taxa de renderização interativa e de uma latência de resposta ao usuário limitada.

Por ter a forma de uma linha de produção, o sistema possui uma taxa de renderização limitada pela velocidade com a qual os quadros são renderizados. O tempo necessário para completar a renderização de um quadro será limitado pelo tempo gasto pelo processador mais lento. No caso da partição do trabalho não estar bem balanceada, a ausência de trabalho decorrente da espera pelo término da tarefa do processador mais lento fará com que os processadores restantes fiquem ociosos, desperdiçando recursos computacionais preciosos que poderiam ser utilizados em outras tarefas. Isso foi ilustrado na Figura 2.7, que se encontra no Capítulo 2.

A latência de resposta ao usuário pode ser medida como o tempo que decorre entre uma ação do usuário e exibição de sua resposta na tela. Em uma aplicação gráfica, uma ação do usuário que ocorre após o início da renderização do quadro i só terá sua resposta exibida no quadro $i + 1$. Medimos portanto a latência de resposta como sendo o tempo que decorre entre esses dois eventos. Como esse tempo é dependente do tempo gasto em todos os estágios da linha de produção, é necessário diminuir o tempo gasto em todos os estágios. Isso é feito reduzindo o tempo gasto na renderização dos quadros, no envio dos *framebuffers* pela rede e no estágio de composição. Para isso, são tomadas medidas tais como a compressão dos *framebuffers*, o envio dos *framebuffers* de um quadro por vez e a limitação do número de quadros sendo renderizados ao mesmo tempo. Essa limitação é feita pelo fato de que em nada adianta sobrecarregar a linha de produção com novos quadros a serem renderizados quando ainda existem quadros sendo processados. Isso somente aumenta a demora na resposta da ação do usuário.

4.2 Algoritmo de Balanceamento Proposto

Um algoritmo eficiente para a partição da tela em ladrilhos deve ser especificado para atender a diferentes objetivos:

- balancear a carga entre os nós de renderização;
- minimizar as interseções de primitivas com as bordas dos ladrilhos, assim evitando o processamento geométrico redundante de primitivas;
- ser geral, se possível não utilizando informações do modelo em questão;
- ser utilizável com cenas dinâmicas;

– ser utilizável em aplicações que possuem gargalo tanto no estágio da geometria quanto no estágio da rasterização;
– ser rapidamente executado, não impondo penalidades adicionais ao sistema de renderização.

Esses objetivos são geralmente conflitantes e uma solução ótima para atender a todos eles pode não ser viável. Por essa razão, os pesquisadores tendem a procurar bons métodos heurísticos para atender a alguns desses objetivos.

O algoritmo aqui proposto visa utilizar a coerência existente entre os quadros renderizados e procura balancear a carga entre os ladrilhos utilizando o tempo gasto para renderizá-los em quadros anteriores, sendo portanto um algoritmo adaptativo. A partição não se baseia no modelo 3D; não obstante, mostraremos nos resultados experimentais que as interseções entre as primitivas e as bordas dos ladrilhos se encontram sob limites aceitáveis.

As vantagens desse algoritmo são a simplicidade de sua implementação e seu tempo de execução, que é praticamente irrelevante. Ele possui bom comportamento tanto em aplicações com gargalo no estágio da geometria quanto em aplicações com gargalo no estágio da rasterização — em contraposição à literatura, que até o presente momento só tratou do balanceamento de carga considerando as primitivas geométricas, o que atende apenas a aplicações com gargalo na geometria.

Para este algoritmo, a tela é dividida em N ladrilhos disjuntos, sendo N o número de nós de renderização. No primeiro quadro a tela é dividida igualmente entre os nós de renderização. A partir disso, o algoritmo utiliza os tempos de renderização do quadro anterior para balancear a partição do quadro seguinte.

Assume-se que o esforço total necessário para renderizar um quadro é a soma dos tempos de renderização de todos os ladrilhos. Para cada ladrilho, considera-se que o seu esforço de renderização está distribuído uniformemente por toda sua extensão. Dessa forma, estima-se um esforço por *pixel* para cada ladrilho. Então, os ladrilhos são redimensionados de forma que todos os ladrilhos tenham a mesma carga.

Mueller [11] mostrou que formas naturais de se subdividir a tela incluem fatias horizontais, verticais e formas mais "quadradas". Formas quadradas são preferidas por minimizar o perímetro dos ladrilhos[1], reduzindo portanto a porção de primitivas geométricas cujas projeções intersectam a

[1]Para uma mesma área, o quadrado é a forma retangular com menor perímetro. A forma de menor perímetro possível é o círculo.

borda dos ladrilhos, que serão processadas redundantemente em todos os ladrilhos em que as primitivas aparecem.

Consideremos primeiramente que a tela será dividida em fatias horizontais, com cada fatia representando um ladrilho. O algoritmo lê o tempo de renderização de cada ladrilho e move as arestas horizontais, somando o esforço de cada *scanline*[2] até que se complete o tempo de renderização esperado por ladrilho, que é a soma dos tempos de renderização de todos os ladrilhos dividido pelo número de ladrilhos. O esforço de uma *scanline* é obtido fazendo-se a divisão do tempo de renderização do ladrilho onde se encontra a *scanline* pela altura do ladrilho. Este procedimento está ilustrado na Figura 4.1

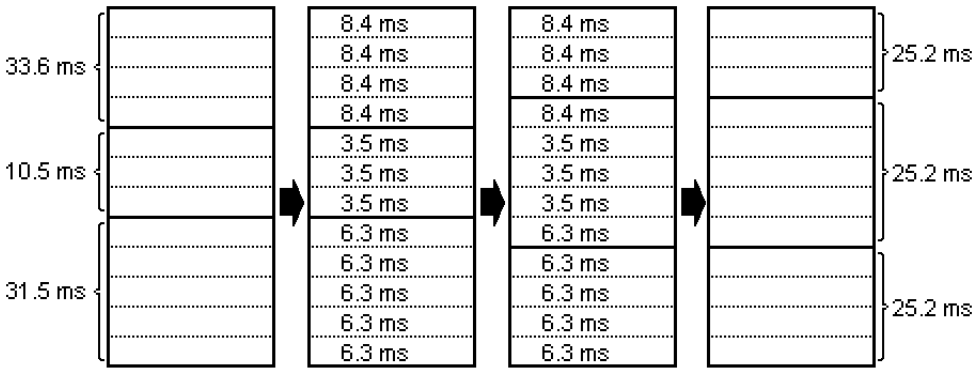

Figura 4.1: Procedimento para o balanceamento de fatias horizontais.

Na tentativa de obter ladrilhos com perímetro menor, o algoritmo primeiramente divide a tela em fatias horizontais e posteriormente divide cada fatia horizontal em fatias verticais, de forma a ter o número total de ladrilhos igual ao número total de nós de renderização. O número de fatias horizontais e o número de ladrilhos por fatia é fixo em funcão do número de nós de renderização. A Figura 4.2 ilustra como é feita a divisão para números diferentes de ladrilhos. Apesar da topologia utilizada ser fixa, ela pode ser definida de forma diferente, dividindo a tela primeiramente em fatias verticais, por exemplo.

Com essa subdivisão da tela, primeiramente o algoritmo reposiciona as arestas das fatias horizontais com o algoritmo descrito acima, considerando o esforço somado dos ladrilhos presentes na fatia para decidir quando parar de adicionar *scanlines*. Então, repete-se o mesmo procedimento para balancear cada fatia horizontal, desta vez somando colunas de *pixels* até que o tempo de renderização médio por ladrilho seja alcançado.

[2] *Scanlines* são linhas de *pixels*.

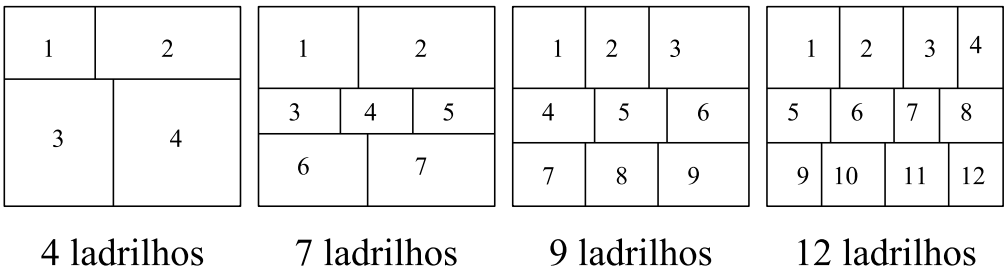

Figura 4.2: Divisão em ladrilhos para números diferentes de nós de renderização.

Esse procedimento pode ser executado em um tempo negligenciável, pois como o esforço por *scanline* ou por coluna de *pixels* é constante por ladrilho, pode ser adicionado um número de *scanlines* ou colunas de *pixels* para alcançar o tempo de renderização esperado para cada fatia horizontal ou cada ladrilho, respectivamente. Isso faz com que o algoritmo tenha complexidade proporcional ao número de ladrilhos, que é sempre um número pequeno.

Pelo fato de o algoritmo não ser baseado no número de primitivas ou em outro tipo de informação derivado do modelo 3D, ele pode ser diretamente utilizado em uma vasta gama de aplicações, que vão desde aplicações com modelos descritos por primitivas geométricas até aplicações que utilizam renderização volumétrica. Usar os tempos de renderização do quadro anterior não representa um problema devido à coerência entre quadros consecutivos. Estes possuem distribuições de esforço similares, mesmo em cenas dinâmicas.

Em contrapartida, o algoritmo supõe que o esforço de renderização é constante em um ladrilho, o que pode ser uma suposição grosseira em muitos casos, especialmente quando o número de nós de renderização é pequeno, resultando em ladrilhos que abrangem boa parte da área da tela. Na tentativa de compensar essa limitação, utilizou-se a coerência espacial, executando o algoritmo de balanceamento apenas para guiar o movimento das arestas dos ladrilhos. Assim que um quadro é completamente renderizado, o algoritmo de balanceamento é executado, porém em vez de mover as arestas da partição para as posições calculadas pelo algoritmo é utilizado o resultado do algoritmo para aplicar velocidades às arestas dos ladrilhos. A posição final de cada aresta é calculada como uma combinação linear entre a posição calculada pelo algoritmo de balanceamento e a posição da aresta no quadro anterior. Nos experimentos realizados, o coeficiente da combinação linear é calculado assumindo que, se a partição balanceada se mantiver a mesma, as

arestas da partição utilizada levariam 200 milisegundos para alcançá-la, o que pode durar de 2 a 10 quadros em aplicações interativas. Como a partição balanceada muda a cada quadro, as velocidades das arestas são reavaliadas e perseguem uma partição diferente.

No cálculo do balanceamento de cada fatia horizontal, cada coluna de *pixels* pode ter *pixels* de diferentes ladrilhos do quadro utilizado como base para o algoritmo. O algoritmo poderia levar isso em consideração ao calcular os tempos de renderização por coluna, mas como o uso de uma distribuição uniforme de esforço por ladrilho já é uma suposição grosseira, o tempo de renderização por coluna foi considerado como o tempo de renderização do ladrilho correspondente no quadro anterior dividido pela sua largura. Isso simplifica a implementação e produz resultados práticos similares.

4.3 Estratégia de Distribuição de Ladrilhos

Devido ao uso de um algoritmo de balanceamento de carga heurístico, nunca será possível calcular uma partição perfeitamente balanceada da tela. Conforme foi ilustrado na Figura 2.7, o desbalanceamento de carga é o maior causador de ociosidade dentre as GPUs do sistema, desperdiçando recursos gráficos que poderiam ser utilizados para melhorar o desempenho da visualização.

Watson et al. [12] demonstraram como as flutuações na taxa de renderização podem degradar o desempenho de aplicações interativas, especialmente quando essa taxa é baixa. Através de seus experimentos, eles concluíram que, para taxas de renderização acima de 20 Hz, variações de até 40% nessas taxas não afetam significativamente o desempenho do usuário em tarefas interativas. No entanto, em aplicações que apresentam necessidade de consistência nos tempos de renderização, as variações destes devem ser mantidas abaixo de 10%.

Propõe-se aqui um algoritmo de distribuição de esforço que visa maximizar o uso dos recursos gráficos dos nós de renderização e reduzir as flutuações nas taxas de renderização do sistema.

Consideremos primeiramente a ausência de um algoritmo de balanceamento. No caso de um sistema com ordenação no início, se mantivermos os ladrilhos com suas dimensões originais (subdividindo a tela em regiões de mesma área, por exemplo), devido à coerência entre os quadros, esperamos que o tempo de renderização de cada ladrilho se mantenha similar em quadros consecutivos. Podemos tirar vantagem de tal coerência fazendo

um rodízio na distribuição de ladrilhos aos nós de renderização. Assim que o nó escravo mais rápido termine sua tarefa de renderização atual, o nó mestre pode lhe requisitar a renderização de um novo quadro. Para melhorar o desempenho geral da aplicação, deve ser requisitada a renderização do ladrilho mais pesado ao nó escravo mais rápido. O ladrilho mais pesado é escolhido com base nos tempos de renderização do quadro anterior, pois podemos supor que os tempos de renderização dos ladrilhos de um quadro futuro serão coerentes com os do quadro anterior. Quando outro nó escravo termina a sua tarefa de renderização, ele recebe o segundo ladrilho mais pesado do quadro anterior, e assim por diante.

Ao se fazer a distribuição de ladrilhos dessa forma, consegue-se um desempenho geral da aplicação próximo ao que seria alcançado se a partição de trabalho estivesse perfeitamente balanceada, ao passo que cada nó se mantém processando tarefas de renderização ininterruptamente. Isso está ilustrado na Figura 4.3.

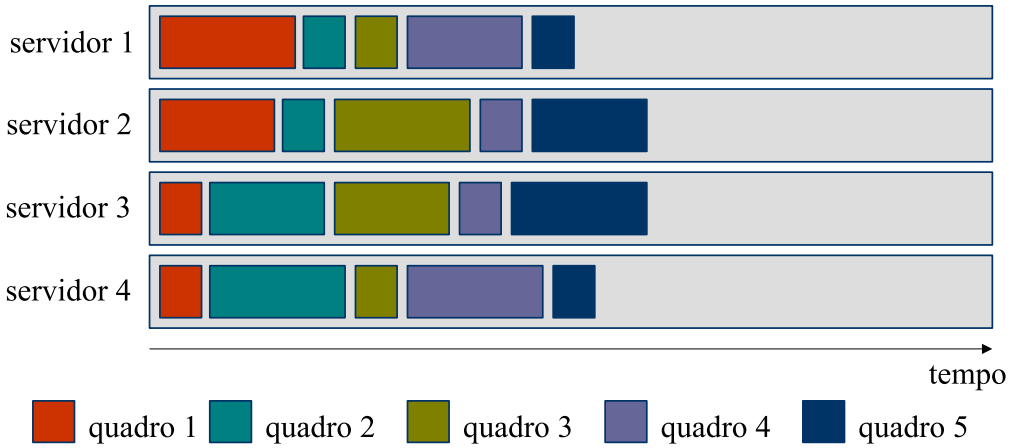

Figura 4.3: Renderização feita com a técnica de distribuição de ladrilhos. A aplicação possui taxa de renderização similar à alcançada com uma partição perfeitamente balanceada.

Entretanto, para partições severamente desbalanceadas, essa estratégia pode causar uma latência de resposta proibitiva, o que degradaria consideravelmente a interatividade do usuário com a aplicação. Essa situação está ilustrada na Figura 4.4. Ao combinar um algoritmo de balanceamento com a estratégia de distribuição de ladrilhos, é possível alcançar um melhor desempenho, mantendo a latência de resposta ao usuário sob limites aceitáveis. Como o algoritmo de balanceamento aplica velocidades às arestas da partição utilizada baseado na partição balanceada em vez de utilizar logo a partição balanceada, é preservada a coerência na ordenação

dos tempos de renderização, o que é essencial para que a distribuição de ladrilhos seja eficaz.

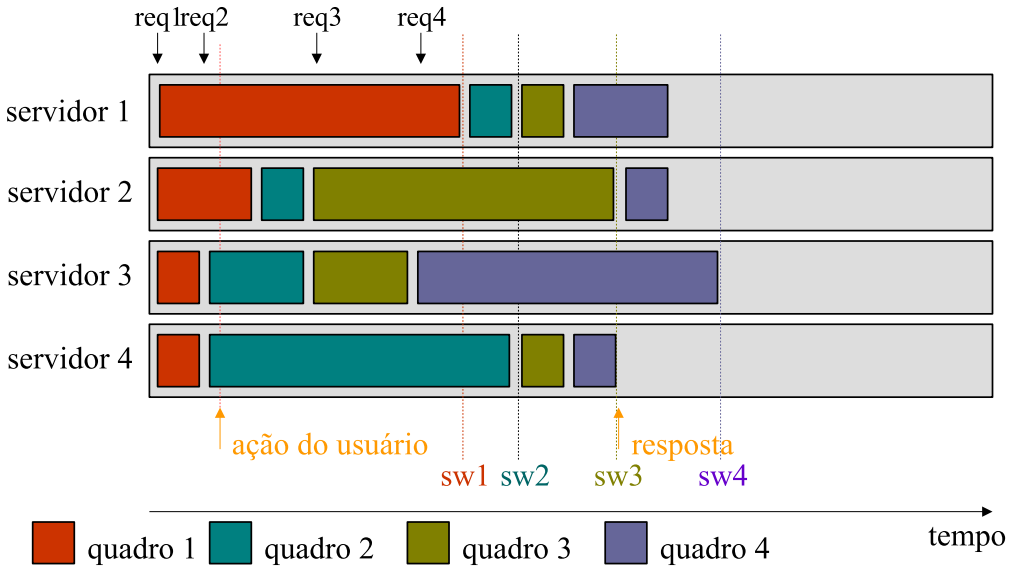

Figura 4.4: Latência de resposta ao usuário com partições severamente desbalanceadas. A ação do usuário só será enviada aos servidores na requisição de renderização do quadro 3, e aparecerá na tela apenas após a troca de *buffers* do quadro 3 (*SwapBuffers*).

Outra vantagem de se utilizar essa estratégia é a suavização das taxas de renderização da aplicação. No caso da aplicação utilizar apenas o algoritmo de balanceamento, este não vai ser capaz de calcular uma partição perfeitamente balanceada, pois o algoritmo é apenas uma heurística. O tempo de renderização do ladrilho mais pesado dessa partição, o qual irá comandar a taxa de renderização da aplicação, sofrerá variações, já que o algoritmo de balanceamento não é perfeito. Se a aplicação utilizar a estratégia de distribuição de ladrilhos, a taxa de renderização da aplicação será comandada pela média dos esforços — como se a partição fosse perfeitamente balanceada. Essa média é proporcional ao esforço total para se renderizar a cena, que varia suavemente à medida que o usuário interage com a aplicação. É esperado portanto que a taxa de renderização seja suavizada com o uso dessa estratégia, melhorando a interação do usuário com a aplicação.

É importante mencionar que a estratégia de distribuição de ladrilhos assume que está sendo utilizado um agrupamento homogêneo de PCs [3], pois espera-se que um dado ladrilho leve o mesmo tempo para ser renderizado em qualquer nó de renderização. Além disso, essa estratégia não é adequada

[3] Agrupamentos cujos nós de renderização possuem igual poder de processamento.

para aplicações que realizam gerenciamento de memória secundária (*out-of-core*), pois cada nó está continuamente renderizando diferentes regiões da tela, ao passo que os algoritmos que realizam tal gerenciamento esperam coerência no que está sendo renderizado a cada quadro.

4.4 Algoritmo de Partição para a Arquitetura Híbrida

No Capítulo 2 foram apresentados os trabalhos relacionados às arquiteturas com ordenação no início e no fim. Na tentativa de combinar as vantagens e atenuar as desvantagens das duas arquiteturas, Samanta et al. [15] apresentaram uma arquitetura híbrida, na qual é feita a partição do modelo 3D e da imagem 2D entre os processadores.

Na arquitetura híbrida é eliminada a grande desvantagem da arquitetura com ordenação no início: o processamento geométrico redundante causado pelas interseções das primitivas geométricas com múltiplos ladrilhos. Na arquitetura com ordenação no fim, a divisão do modelo entre os nós de renderização permite que cada primitiva seja desenhada apenas uma vez, porém torna necessária a transferência de um número de *buffers* de cor e informações de visibilidade proporcional ao número de processadores do agrupamento. A grande largura de banda utilizada por essa arquitetura torna-se sua maior limitação. É possível reduzir a quantidade de informações transmitidas pela rede particionando-se o modelo de forma a minimizar a região de interseção entre as áreas da imagem que cada parte do modelo desenha. Isso é feito considerando-se a distribuição dos objetos da cena no espaço da tela. A redução na quantidade de informações transmitidas está ilustrada na Figura 4.5.

A técnica apresentada por Samanta et al. [15] obteve bons resultados em termos de escalabilidade com os modelos utilizados. No entanto, o algoritmo utilizado apresenta um possível gargalo no estágio de partição do modelo entre os processadores, pois a sua complexidade é proporcional à complexidade do modelo.

Na tentativa de desenvolver um sistema híbrido sem essa limitação, apresentamos um método em que a partição do modelo é feita nos servidores de forma implícita, o que a paraleliza entre os nós de renderização.

O algoritmo aqui proposto baseia-se na divisão da tela em ladrilhos disjuntos, assim como na arquitetura com ordenação no início. No entanto, na tentativa de minimizar o número de primitivas processadas de forma redundante, cada objeto da cena é implicitamente atribuído a um único nó

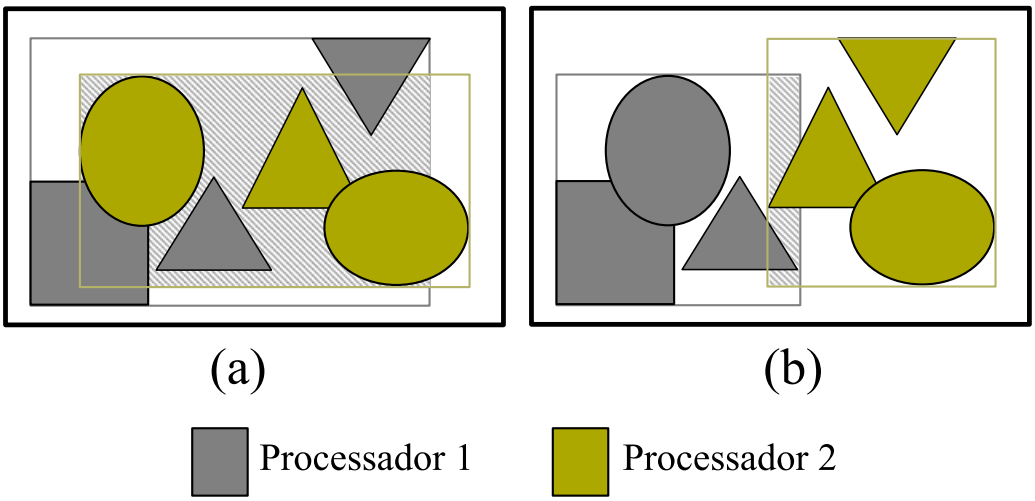

Figura 4.5: A influência da distribuição espacial dos objetos na quantidade de dados a serem transmitidos: (a) agrupamento aleatório; (b) agrupamento espacial.

de renderização. Essa atribuição é feita com base na posição do centro do volume envolvente[4] do objeto. O seguinte critério é adotado com relação a cada objeto em cada nó de renderização:

- os objetos que se encontram totalmente fora do volume de visão são descartados;
- os objetos que se encontram totalmente dentro do volume de visão são desenhados;
- os objetos que intersectam algum plano do volume de visão são desenhados se o centro do seu volume envolvente estiver dentro do volume de visão.

Desta forma, o algoritmo de partição pode tirar proveito das acelerações usuais do algoritmo de descarte pelo volume de visão em cada nó de renderização. No caso do objeto intersectar algum plano do volume de visão, ele será desenhado em apenas um servidor. O *Z-Buffer* da área que esse objeto ocupa nos outros ladrilhos com os quais ele tem interseção deve ser lido e enviado ao nó responsável pela composição, para que este seja capaz de resolver a visibilidade nessa área. Isso está melhor ilustrado na Figura 4.6.

Existem três maneiras de calcular qual área deve ser lida com informações de visibilidade, dados os objetos que intersectam as bordas do ladrilho:

[4] *Volume envolvente* é um volume de forma geométrica simples que envolve toda a geometria de um objeto ou grupo de objetos. Ele simplifica operações como detecção de colisão e interseção do objeto com os planos do volume de visão.

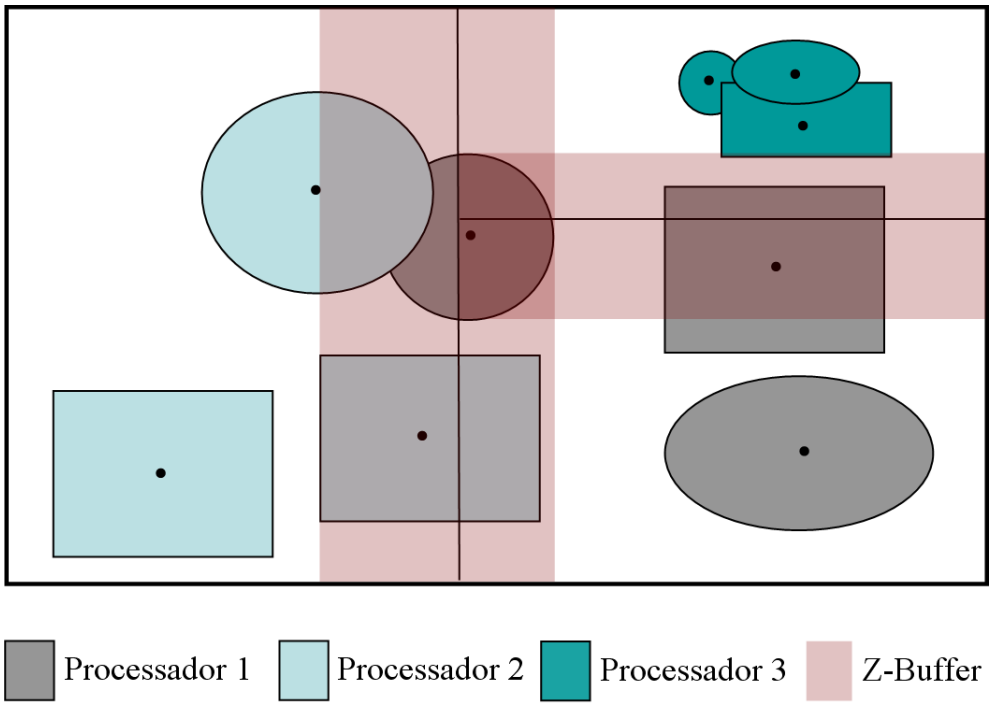

Figura 4.6: Configuração com 3 nós de renderização. Em vermelho, a área que requer envio de *Z-Buffer*.

- cálculo do retângulo mínimo que envolve todos os objetos;
- cálculo da largura do retângulo que envolve todos os objetos e tem a mesma altura da tela;
- uso de uma faixa fixa.

Essas maneiras estão ilustradas na Figura 4.7. Apesar das duas primeiras calcularem uma área menor a ser lida, elas requerem uma quantidade de cálculo proporcional ao número de objetos, o que pode ser muito custoso dependendo do modelo. O uso de uma faixa fixa para a leitura foi escolhido para o protótipo implementado.

Os objetos que se estendem além das faixas determinadas são desenhados em todos os ladrilhos que intersectam, como na arquitetura com ordenação no início. Isso evita que o algoritmo tenha que ler e transmitir faixas muito grandes do *Z-Buffer* devido a objetos que têm uma vasta projeção no espaço de tela.

No nó mestre utilizou-se o mesmo algoritmo de balanceamento de carga empregado no sistema com ordenação no início, pois o algoritmo é geral e serve para balancear o esforço entre os processadores. Desta forma, o esforço feito na partição dos objetos visíveis da cena é distribuído entre os processadores.

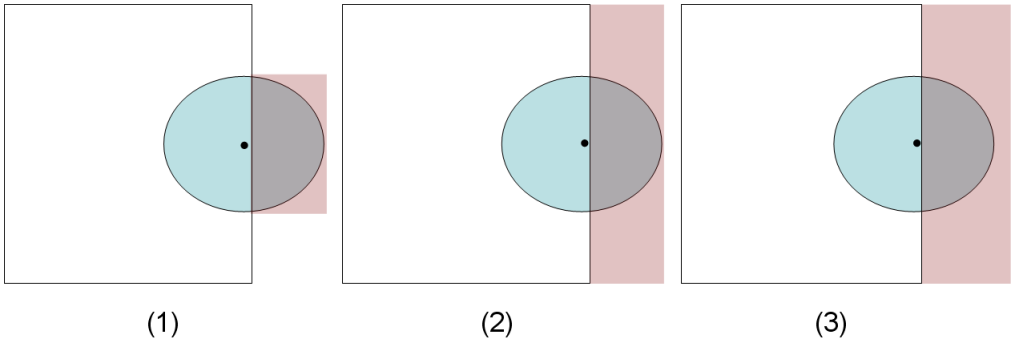

Figura 4.7: Maneiras possíveis de se calcular a área do *Z-Buffer* a ser lida: (1) cálculo do retângulo mínimo que envolve todos os objetos; (2) cálculo da largura do retângulo que envolve todos os objetos e tem a mesma altura que a tela; (3) uso de uma faixa fixa.